Süsteemprogrammeerimine keeles C

Loeng 8

Protsesside loomine ja hävitamine

- UNIX pakub nelja süsteemikäsku protsesside loomiseks, lõpetamiseks ja nende lõpetamise ootamiseks
  - exec() perekond
  - fork()
  - wait()
  - exit()

Protsessi mällu laadimine

- *Binary executable* koosneb harilikult päisest, (programmi)tekstist, andmetest, andmeteisaldusinfost ja sümboltabelist. Päist, sümboltabelit ja andmeteisaldusinfot kasutatakse korra ja siis visatakse minema, tekst ja info aga jääb mällu.

Executable file	Process memory
HEADER	TEXT
TEXT (program)	DATA
DATA (initialized)	(BSS)
BSS (=uninitialized data)	free mem
RELOCATION	STACK
SYMBOL TABLE (can be stripped)	USER BLOCK (in kernel adr space)

exec() käsuperekond

- Exec() käsud laevad käivitatava faili (*binary* executable) mällu protsessiks. Süntaks on järgmine:

```
extern char **environ;
int execl( const char *path, const char* arg, ...);
int execv (const char *path,  char *const argv[]);
int execle(const char *path,
          const char *arg, ..., char * const envp[]);
```

  - execl: täielik failinimi, var. argumendid charidena
  - execv: täielik failinimi, argumendid massiivis
  - execle: täielik failinimi, var. argumendid charidena, keskkond (*environment*)
- Mis on **keskkond**?

getenv()

- Getenv annab viite keskkonnamuutujate stringi

```
#include <stdlib.h>
char *getenv(const char *name);
```

- vt ka getenv() putenv()

exec() perekond (2)

- Tegeliku töö teeb ära see funktsioon, mida teised kutsuvad:

```
int execve (const char *filename,
                char *const argv [],
                   char *const envp[]);
```

  - täielik failinimi, argumendimassiiv, keskkonnamassiiv

```
int execlp( const char *file, const char *arg,
                ...);
int execvp( const char *file, char *const argv[]);
```

  - PATH otsing, var. argumendid & PATH + argumentmassiiv

```
char* getenv(const char *name);
```

exec() perekond (näpunäited)

- Esimene argument peab olema käivitatava programmi **nimi**. "Päris" argumente sinna panna on vale.
- Argumendi ja keskkonnapointerite massiiv **peab** lõppema NULL pointeriga
- Kui te keskkonda ei edasta, antakse edasi jooksva programmi keskkond: st *mingi* keskkond on programmil igal juhul olemas
- **exec ei lõpeta.** Kontroll antakse programmile, mis sa argumendiks andsid ja sinu programmi enam ei täideta. Kui exec tagastab väärtuse, tekkis viga.

Näide (kataloogi sisu näitamine)

```
#include <stdio.h>
main()
{
int err;
if (err = execl("/bin/ls", "ls", "-l", "/etc",
NULL))
    printf("Err=%d\n",err);
else
    printf("Mind ei trükita kunagi välja\n");
}
```

fork()

- Kuna exec asendab olemasoleva protsessi, peame kuidagi saama ka teha uusi. Uue jaoks on käsk:

```
pid_t fork()     /* pid_t is an int */
```

- Loob lapsprotsessi uue PID-ga. Kutsume välja ühe korra, tagastume kaks korda.
- Uus protsess on väljakutsuva (*parent*) täpne koopia.
- Tagastab lapse puhul 0 ja vanema puhul PID
- Vanem saab -1 ja errno, kui ei õnnestu
- Laps saab vanema kasutaja tegeliku ja kehtiva ID, grupi tegeliku ja kehtiva ID, keskkonna, avatud failideskriptorid (kopeeritakse) jne...

wait()

- fork() ei oota, et laps oma töö lõpetaks

```
pid_t wait(int *status);
```

- status on pointer kohale, kuhu süsteem võiks salvestada lapse poolt väljumisel tagastatud väärtuse
- Tagastusväärtuseks on lõpnud lapse PID
- Ootab kuni laps on lõpetanud
- Kui laps lõpetab, muutub ta **zombiks**: kernel jätab osa tema kohta käivast infost meelde ja koristab ülejäänu ära. Kui vanem lõpetab lapse vastu huvi tundmata, teeb init protsess zombikoristuse ise ära.

waitpid()

- Analoogne funktsioon, aga saate öelda millise lapsprotsessi järel oodatakse

Shelli näide

shell.c

system()

- exec() asemel on tihti kiirem kasutada kestakontekstis käivitumise funktsiooni system()
- Käivitab programmi /bin/sh -c käsu abil

exit()

```
void exit(int status);
int atexit(void (*func)(void));
```

- exit() lõpetab protsessi töö viisakalt ära. Vanemale tagastatakse väljumisväärtus, kõik atexit() funktsioonid kutsutakse välja registreerimisele vastupidises järjekorras ning kõik avatud stream'id *flush*itakse ning suletakse
- exit() ei tagastu
- atexit() registreerib argumendina antud funktsiooni exit() käsu puhul käivitamiseks
- Viisakas lõpp on kas exit() abil või return main() funktsioonist

atexit() näide

```
#include <stdio.h>

void final_wish(void);

main() {
     atexit(final_wish);
     printf("Hello World!\n");
}

void final_wish(void) {
   printf("Wish I could have lived forever\n");

}
```

Deemonid

- Deemonid (Daemon) on serveriprotsessid, mis jooksevad taustal. Näiteks võid oma FTP serveri deemonina programmeerida.

```
void run_server(void);
main()  {
int pid;
if ((pid = fork()) < 0) {
        perror("server process does not fork");
        exit(1);
   }
if (pid==0)
        run_server();
else {
       printf("FTP daemon successfully launched, pid=%d\n",
                        pid);
       exit(0);
      }
}
```

Protsessi omanik ja grupp

- *Real id* ütleb, kes sa tegelikult oled, *Effective id* ütleb, mis õigused on sul failidele ja seadmetele.

```
uid_t getuid(void);      /* uid_t on int */
uid_t geteuid(void);
```

- Tegelik id vastab väljakutsunud protsessi UIDle
- Kehtiv id vastab setuid biti poolt määratud UIDle

```
gid_t getgid(void);
gid_t getegid(void)
```

- Tegelik grupi id vastab väljakutsuva protsessi GIDle.
- Kehtiv grupi id vastab set ID biti poolt määratule

Protsessi identiteet

```
pid_t getpid(void);
pid_t getppid(void);
```

- getpid() tagastab parajasti jooksva protsessi PID (seda kasutatakse tihti nt ajutiste failide unikaalsete nimede loomisel)
- getppid() tagastab parajasti jooksva protsessi vanemprotsessi PID
- init on protsess nr 1.
- Kõik peale mõne üksiku kerneliprotssi põlvnevad initist